PROBLEM SOLVING AND NONDETERMINISTIC PROGRAMMING SYSTEMS

William George Kennedy

Library
Naval Postgraduate School
Monterey, Cali   93940

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

PROBLEM SOLVING
AND
NONDETERMINISTIC PROGRAMMING SYSTEMS

by

WILLIAM GEORGE KENNEDY

Thesis Advisor: G. D. Gibbons

June, 1973

T154875

Approved for public release; distribution unlimited.

PROBLEM SOLVING
AND
NONDETERMINISTIC PROGRAMMING SYSTEMS

by

William George Kennedy
Ensign, United States Navy
B. S. United States Naval Academy, 1972

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June, 1973

Library
Naval Postgraduate School
Monterey, California 93940

ABSTRACT

This paper suggests and discusses an implementation of several ideas in the area of problem solving and nondeterministic programming systems. After discussing the history of work in this field, definitions of forward, horizontal, and backwards simple moves are given. With these definitions, the level of a problem is defined to be the maximum number of consecutive backwards moves required to solve the problem. Level zero problems can be solved using forward and horizontal moves only. It is then shown how two methods, the combination of moves and problem reduction, sometimes reduce the level of a problem. The use of Gibbons' Q-size rule to prevent redundancy is shown to sometimes conflict with heuristic search methods. The use of a hashing function to detect redundancy is discussed. The choosing of moves according to their evaluation of move types was used in a program, POPS II, based on Gibbons' POPS, which was shown to be more efficient than previous nondeterministic problem solvers for several problems.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF DRAWINGS

ACKNOWLEDGEMENT

I would like to thank my advisor, Professor Greg Gibbons, for being my heuristic in the search for the end of this thesis. I would also like to thank Ms. Frances Kennedy, my wife, for listening to me and arguing with me and for solving the M + C problem without a backwards move. I also wish to thank Ms. Madeline Cirillo and Ms. Rosary Cirillo for their editorial help to make this thesis possible.

I. INTRODUCTION

To solve a problem using a computer requires the programming of an explicit list of instructions of what the machine is to do under specified circumstances. The computer then does exactly what is stated, not always what is wanted. A computer which would do what is wanted requires some "intelligence" on the part of the computer and the design of such a machine falls in the field of Artificial Intelligence. One area of Artificial Intelligence is problem solving. Problem solving efforts include work being done to design a computer which would solve a problem when it is given only the problem, and not a way to solve it.

This paper deals with the design of problem solving systems using programming languages which are nondeterministic in nature. To understand the following chapters some background is necessary in the terminology and methods of problem solving and on the subject of nondeterministic algorithms.

A. PROBLEM SOLVING

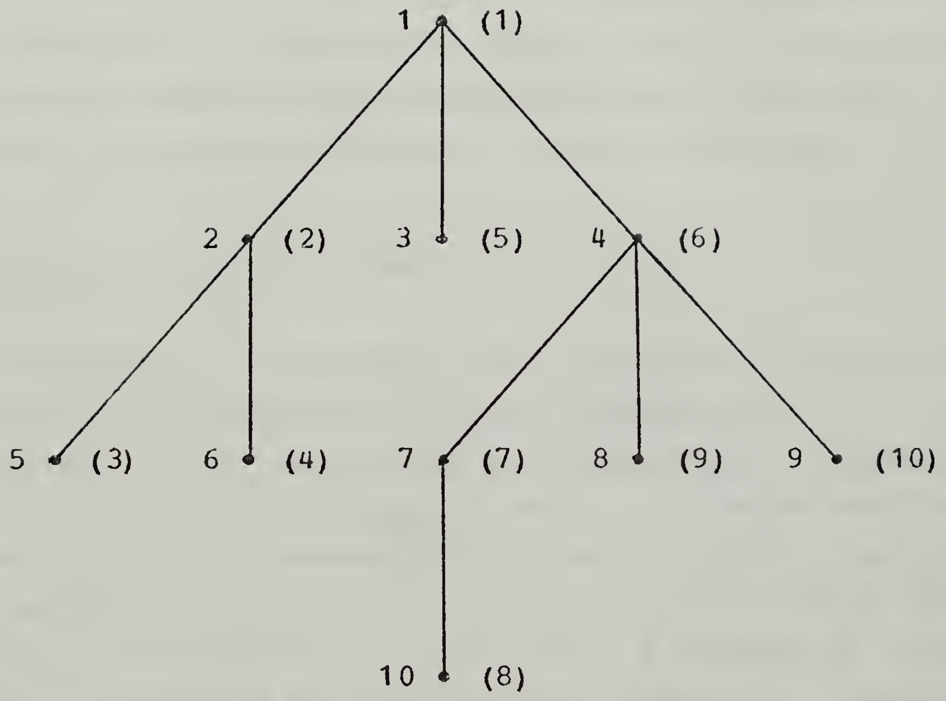
Some problems can be stated in terms of an initial situation, a collection of actions which can be taken, and a goal that is to be reached. A state is a particular situation in terms of variables and their assigned values. The initial situation of the problem is called the initial state and the goal is the goal state. Operators are defined to be actions which transform one state into another state.

The solution is a sequence of operators which transforms the initial state into the goal state. The solving of a problem is therefore the search for this sequence. The search space is the collection of states which can be reached starting with the initial state and applying any series of operators. The search space could be very small or it could be infinite.

Thus solving a problem becomes the search of the space for a path to the goal state. This search can be blind if the method of generating successive states is independent of where the goal is located in the space. If on the other hand, the search is guided in some way using information contained in the problem, the search is heuristic in nature. A heuristic is a method of guiding the search in the general direction of the goal.

Search spaces can be considered to be in the form of mathematical trees [Knuth 1968] where each state is a node in the tree. The search of the space is equivalent to the expansion of the nodes of the tree. There are several blind methods to generate or search the tree, of which two are basic. An example of these methods is shown in Figure 1. Each blind method applies operators using a predetermined ordering. The first method is called a breadth first search. This applies each operator once to the last generated nodes on the tree, producing one node for each operator applied to each node. The second method is called a depth first search. One node is selected and expanded by applying an operator and then the resulting node is expanded in the same manner. When the expansion cannot continue, the method backs up to the immediately preceeding node and applies the next operator.

EXPANSION OF A NODE



The nodes are numbered as they would to be generated in a breadth (depth) first expansion of the tree.

Figure 1

An heuristic search is one where the choice of the next node to expand and/or the next operator to apply is guided by a method dependent on information contained in the problem. This information is usually used to guide the search in some sense "toward" the goal node. The efficiency of a search is defined as the number of nodes in the path to the solution divided by the number of nodes searched. The ideal heuristic is 100% efficient, i.e., it guides the search directly toward the solution without generating any other nodes than those in the path to the solution.

B. NONDETERMINISTIC ALGORITHMS

A deterministic algorithm is a series of instructions whose execution is specified by these instructions and any data entered. This means that its operation is completely defined by its instructions and input. A nondeterministic algorithm is like a deterministic one except its operation is not completely specified. At a branch point in a nondeterministic algorithm, the next instruction is not necessarily prescribed by the data and values of variables at that point. Nondeterministic does not imply probabalistic in nature, but that something outside the algorithms makes the decisions. There are two methods to understand this guidance. One is to assume that at each decision point the correct selection is made through unknown methods. This has been called the Superintelligent Model [Gibbons 1972]. The second way to explain this guidance is that at each nondeterministic decision point several independent machines are set up executing each possible interpretation simultaneously. At the end, if one of the machines reaches the desired result, that machine is taken to have been executing the correct selections and all machines which have failed would be ignored.

Nondeterministic algorithms cannot be executed by computers, but there is a way to convert nondeterministic algorithms into executable deterministic algorithms. A nondeterministic algorithm is equivalent to a nondeterministic finite automaton which is a theoretical machine that has a mathematical definition [Hopcroft 1969]. Since there is a method to convert nondeterministic finite automatons into deterministic ones, there must also be a way to convert nondeterministic algorithms into deterministic algorithms. This is the basis of problem solving systems using computers to execute nondeterministic algorithms.

II. PREVIOUS WORK

A. FLOYD'S SUGGESTION

Robert W. Floyd [Floyd 1967] recognized that nondeterministic algorithms were often an easier and a more natural way of stating problems than deterministic algorithms. The former method of stating problems is not readily available because computers can execute only deterministic algorithms. Thus what was needed was a nondeterministic language and a method to execute programs written in this language. Floyd proposed two changes to a programming language which would produce a nondeterministic language. He then suggested methods to transform algorithms written in this new language into executable deterministic algorithms.

1. A Nondeterministic Language

The changes necessary to produce the nondeterministic language included a new function and the labeling of exits from the program. The value of the function, CHOICE(N), was defined to be some integer in the range from one to N. This function frees the programmer from having to specify the value of a variable and leaves its correct selection up to the compiler or assembler. Exits from the program were to be labeled as SUCCESS or FAILURE. The execution of the program would then be designed to find a way to exit the program through a SUCCESS exit. Floyd stated that these were the minimum changes necessary to obtain a useful nondeterministic programming language. With these changes a problem such as:

```
find X and Y in (1,10)
such that
     X+Y=15
```

could be written in an ALGOL-like language as:

```
BEGIN
     INTEGER X,Y;
     X:=CHOICE(10);
     Y:=CHOICE(10);
     IF (X+Y=15) THEN GO TO SUCCESS;
FAILURE:
SUCCESS:
END
```

2. A Simple Processor

Floyd's implementation of a nondeterministic programming language was discussed in terms of flowcharts which could be readily converted into statements in any language. Floyd outlined transformations that provided a box-by-box conversion of a nondeterministic flowchart to a deterministic one suitable for programming. This conversion preserves the search space described in the original problem statement. Figure 2 shows a transformation of the flowchart for the above problem containing the flavor of Floyd's design. As is clear in the figure, the execution of the transformed program was a search of the space by the enumeration of the values for each CHOICE function. This was done by the sequential assignment of values to each CHOICE function and by backtracking when a failure exit was taken by the transformed code at the next level down in the program. Backtracking was the undoing of some previous work, the changing of the value of a CHOICE function to the

FLOYD'S TRANSFORMATION

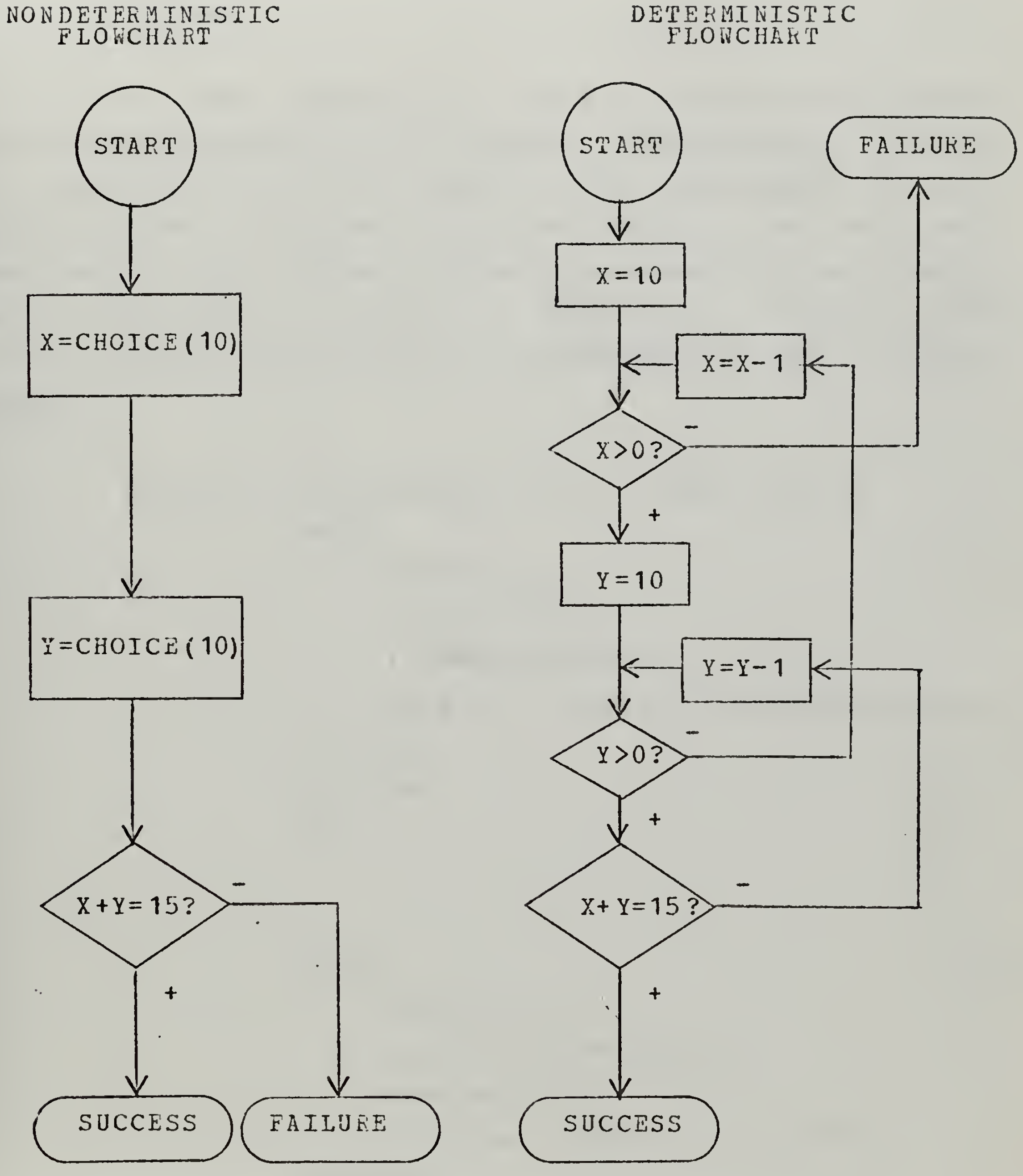


Figure 2

next possible value, and then the re-execution of the same instructions.

3. Comments

The time required to find a solution in Floyd's implementation depends on the problem formulation, because the pattern of the search is determined by the transformations. This means the execution time was very sensitive to the problem statement; no analysis was done on the input program to choose a method of searching the solution space. This point is illustrated in the following example.

Here are two programs for the same problem:

```
BEGIN
     INTEGER X,Y;
     X:=CHOICE(2);
     Y:=CHOICE(1000);
     IF X*Y¬=1 THEN GO TO FAILURE;
SUCCESS:
FAILURE:
END
```

```
BEGIN
     INTEGER X,Y;
     Y:=1001-CHOICE(1000);
     X:=CHOICE(2);
     IF X*Y¬=1 THEN GO TO FAILURE;
SUCCESS:
FAILURE:
END
```

Since the transformations start with the highest value for the CHOICE function and work down, the first program will go

through 2000 trials while the second program goes through 2 trials to achieve the same goal where X=Y=1. Therefore, with this method of implementation, the time required to solve this problem is very sensitive to the problem statement.

The practicality of this method of implementing a nondeterministic programming language is also dependent on the size of the search space. On problems with a small search space, this method could be very reasonable. For these problems, the analysis of the input program done by more advanced systems would be more costly than the simple enumeration done by Floyd's method. Most problems have large, often infinite, search spaces. For such problems, Floyd's transformations may fall far behind more sophisticated implementations of nondeterministic programming languages.

B. FIKES' REF-ARF

Richard E. Fikes [Fikes 1968,1970] produced one of the first systems that implemented Floyd's suggestion. Rather than expand a current programming language, Fikes designed a new, simplified language, REF. The problems written in REF were then interpreted on the ARF interpreter. The interpreter attempted to find a way to get to a successful termination as in Floyd's proposed system. ARF employed constraint satisfaction and heuristic search methods, as explained below, to control this search.

1. The REF Language

An understanding of the REF language is necessary because the systems discussed in this paper use similar languages. REF was designed to be a simple language to

implement, as opposed to an expansion of a general programming language such as FORTRAN, COBOL, or ALGOL. REF's simplicity allowed the major portion of the effort to be spent on the interpreter. Although REF had limitations, it allowed representation of a large range of problems.

The major difference between Floyd's suggested extension of some current language and Fikes' REF was the replacement of the labeling of all exits. This was done with CONDITION(P) statements and the assumption that all exits were successes. Unless the boolean expression, P, in the CONDITION statement could be evaluated to true, the processing would halt with a failure as a result. For example, the code

```
IF (X¬=1) GO TO FAILURE;
```

is replaced in REF by

```
CONDITION (X=1);
```

This way the reaching of the END statement with all the required expressions evaluating to true represented a successful execution.

REF had several other smaller changes and limitations which distinguished it from other programming languages. It was missing real valued functions due to the fundamental limitation of finite sets for ranges. It was also missing numeric operators other than addition and subtraction. It also did not have block structuring and subroutines. Variables were limited to only one dimensional arrays. Fikes referred to each element in an array as a slot. The simple assignment statement was altered from X(1)=3 to SET X(1) TO 3. There was also a similar statement to set the value of an entire vector at one time. Floyd's CHOICE(N) function was directly replaced by SELECT(I,J). The value of this function was defined to be an integer in the range from I to J inclusive. There were also IF and computed GOTO statements as found in other programming

languages. As an illustration, the problem discussed in Floyd's section would have been written in REF as:

```
BEGIN;
   SET X(1) TO SELECT(1,10);
   SET Y(1) TO SELECT(1,10);
   CONDITION X(1)+Y(1)=15;
END;
```

2. The ARF Processor

ARF, the interpreter for ARF, attempted to find a path through the program to the END statement. To do this ARF had to choose values for each SELECT(I,J) function which would define a path through the program and whose values satisfied all the constraints imposed by that path. The major innovation was an improvement on Floyd's automatic enumeration of the possible values. This was accomplished by the assignment of a symbol rather than a number as the value of each SELECT function. Each symbol assigned to a nondeterministic slot had the range I to J associated with it. The symbol was considered to have a constant value but unknown at the time of assignment. This delaying of the actual assignment of values allowed the processor to go on processing without the enumeration of the possible values in hopes that the range of values would be reduced by some later processing. This was done so that if a search of the space had to be done at a later time, it would possibly be a much smaller space. The symbolic assignment of the SELECT function increased the complexity of the interpreter's task. A slot whose value was unknown was not assigned a value until a value for that slot was required in order to interpret a future statement.

If a slot whose value was unknown was used to control branching at some later point in the program, then the next instruction to execute also became unknown. This could occur if one of these slots was the index to a computed GOTO or was in an expression for an IF statement. This fact necessitated the idea of a context which would be used to keep track of the values assigned to the slots. The context was a collection of data which described a point in the interpretation of a program. This data was complete enough that at some later time the context could be used to return the interpreter to this point and begin processing again. A context included the following:

(1) The next instruction
(2) The current values of all slots
(3) The current ranges for symbols associated with SELECT functions
(4) The constraints up to this point

Constraints were boolean expressions. For example, each interpreted CONDITION statement added its expression to the current context as a constraint. The processing of a constraint caused, in come cases, the limiting of ranges for nondeterministic slots. If constraint processing led to the discovery of contradiction, the context was deleted from further analysis.

When ARF interpreted an IF statement whose boolean expression was nondeterministic, two contexts were generated with different constraints and next instructions. In one context, the boolean expression in the IF statement was added as a constraint and the next instruction was the instruction which would follow if the expression had been true. The other context contained the negation cf that

expression and the next instruction was the one following the IF statement. Similarly, a nondeterministic computed GOTO statement generated one context for each possible value of the index. Assignment statements could also generate several contexts if the variable to be assigned a value was undeterminable due to a nondeterministic index into the vector. This generated one context for each value in the range of the index variable.

When the END statement was reached, the context had to be evaluated to see if there were a set of values for the SELECT's which satisfied all the constraints. If there were such values, a solution had been found. The context at the end represented a constraint satisfaction problem for the processor, since there was a problem involved with finding satisfactory values for each SELECT. How REF-ARF conducted the heuristic search for a path through the program will be discussed before discussing REF-ARF's constraint satisfaction techniques.

3. Heuristic Search Methods

The heuristic search mechanism conducted a search where the collection of contexts was the search space and the REF statements were the operators. The search started with the null, or empty, context at the BEGIN. It then went into a loop where it selected a context from the saved contexts and applied the operator which was the next instruction indicated by the context. If the END statement had been reached, the constraint satisfaction routines were applied. Then if a solution was found the interpreter halted. If not, any contexts created were saved and control went back to the top of the loop to choose another context and go through the loop again.

This searching was controlled by three methods, of which two attempted to cut down the size of the search space and the third controlled the order in which contexts were selected. The reasons used for this reduction were that a context was equivalent to a previous context or that there was a contradiction within the context. The third method to control the searching was to choose the context which was the "closest" to the END statement.

When a new context was tested, it was compared against previous contexts to check for redundancy. Since identical contexts were rarely found, this was a test for equivalence between two contexts. Two contexts were considered equivalent if the ranges of variables, created by the SELECT functions, in the new context were contained within the ranges of the saved context. This test was only applied between contexts which had no unsatisfied constraints. It was hoped that this test would detect obvious looping and also be fairly inexpensive to use.

Second, a compromise was reached between processing constraints after every instruction, and doing this processing only upon reaching the END statement. The result was that before processing an instruction which generated multiple contexts, such as an IF, computed GOTO, or SET statement, the constraints of the current context were tested for a contradiction. The discovery of a contradiction eliminated the context from consideration and eliminated the multiple contexts which would have been generated. By reducing the ranges of variables, this programming would sometimes eliminate some futile branches of the search space as well.

Last, a heuristic was used to guide the selection of which context to work on next. The context chosen was that

context which was "closest" to the END statement. This "closeness" was determined by a structural analysis of the program at the start. This analysis associated with each instruction the number of instructions in the shortest possible path from that instruction to the END statement. This made it possible to choose the "closest" context to the END by choosing the context with the minimum number of possible instructions to the END.

4. Constraint Satisfaction Methods

When the processor reached the END statement and there were constraints in the context, the constraint satisfaction routines were called upon to discover the solution or to prove that there was no solution. These routines were designed to find values for the SELECTs which would satisfy all the remaining constraints. If there was no set of values for these variables which would satisfy all the constraints, then there was no solution within this context. To prove there was no possible set of values, the processing did not simply enumerate the possible values, but attempted to find contradictions within the constraints. In order to satisfy the constraints, or to prove there was no set of values possible, this processing was conducted in two modes; constraint manipulation, and the assignment of values to slots.

Constraint manipulation was the simplification and combining of constraints in the context in order to find a contradiction or to reduce the ranges of the values for variables. In order to achieve these results, several formula manipulation routines were used. The discovery of a contradiction eliminated a context from further analysis. The limiting of the ranges for the variables would greatly reduce the searching necessary later. For instance, if one of the constraints was X(1)=3, then it is obviously

unnecessary to try values for X(1) other than 3. Thus, X(1) would be assigned the value 3.

REF-ARF used several types of constraint manipulation routines to achieve these results. It used routines to simplify and standardize simple expressions, to make deductions from single expressions, and to make deductions from combinations of expressions. If a range for a variable was (1,10), and a constraint was X(1)<2, then the range could be reduced to (1). This had the effect of assigning 1 to X(1). If there were two constraints, such as X(2)<5 and X(2)=7, then a contradiction would be discovered and the context deleted.

Along with these routines was a routine which assigned values to nondeterministic variables. This routine made the first assignments to the variable with the most restricting constraints. Fikes ordered the logical operators by how limiting he felt they were. Equality was considered to be the most limiting and the OR operator was the least. The strategy was to exhaust the possibilties for the most limited variables first by enumeration in hopes that their values would be found with the least effort or that a contradiction would be found faster than for other variables.

The routines described above were grouped together into a constraint satisfaction section. When working on the context at the END statement, the constraints were processed to initially simplify the problem. Then an assignment was made and the constraints processed again. If a contradiction was discovered, the process recorded a failure. If not, and there were constraints still remaining, this process was called again recursively. When a recursive call returned a failure, another assignment was made and the process called again. If a point was reached

where there were no constraints remaining, a success was returned. The other possibility was that there were no more values to try assigning and a failure resulted. Fikes stated that this combination of constraint manipulation and assignments of values to variables was more powerful than either used seperately.

5. Comments

A problem could basically be stated either as a constraint satisfaction problem or as a heuristic search problem. On problem statements which were basically constraint satisfaction problems, REF-ARF performed quite well due to its constraint manipulation methods. On problems stated basically as heuristic search problems, the system did not do as well. This was due to its rather weak heuristic guidance of the search. A more complete description of the performance of REF-ARF on these types of problems is found in [Fikes 1970].

C. GIBBONS' POPS

Gregory D. Gibbons [Gibbons 1972,1973] designed a problem solver in LISP [Prichard 1969, Bolce 1967] incorporating some of the REF-ARF constraint manipulation techniques and new heuristic search methods. The REF language was modified slightly and called PSL.

1. The PSL Language

PSL, a problem statement language, was basically REF except it was LISP oriented and a few of the REF statements were changed. The SELECT function was modified to have one argument which was the name given a range of values. This name was established in a RANGE statement. The RANGE

statement attached a name to a set, which need not have numeric elements. Again, using the same problem, this is demonstrated in the following example:

```
( (RANGE A (1 2 3 4 5 6 7 8 9 10))
  (*SET (X 1) TO (SELECT A ))
  (*SET (Y 1) TO (SELECT A ))
  (CONDITION (= (+ (X 1) (Y 1)) 15))
  (END) )
```

The computed GOTO was replaced by a GOTOL, go to list, which had one argument which was a list of labels for which there was no index. The next instruction executed was any one of the statements whose label was in the list. This was intended to be a nondeterministic branch. The simple GOTO and IF statements remained basically unchanged.

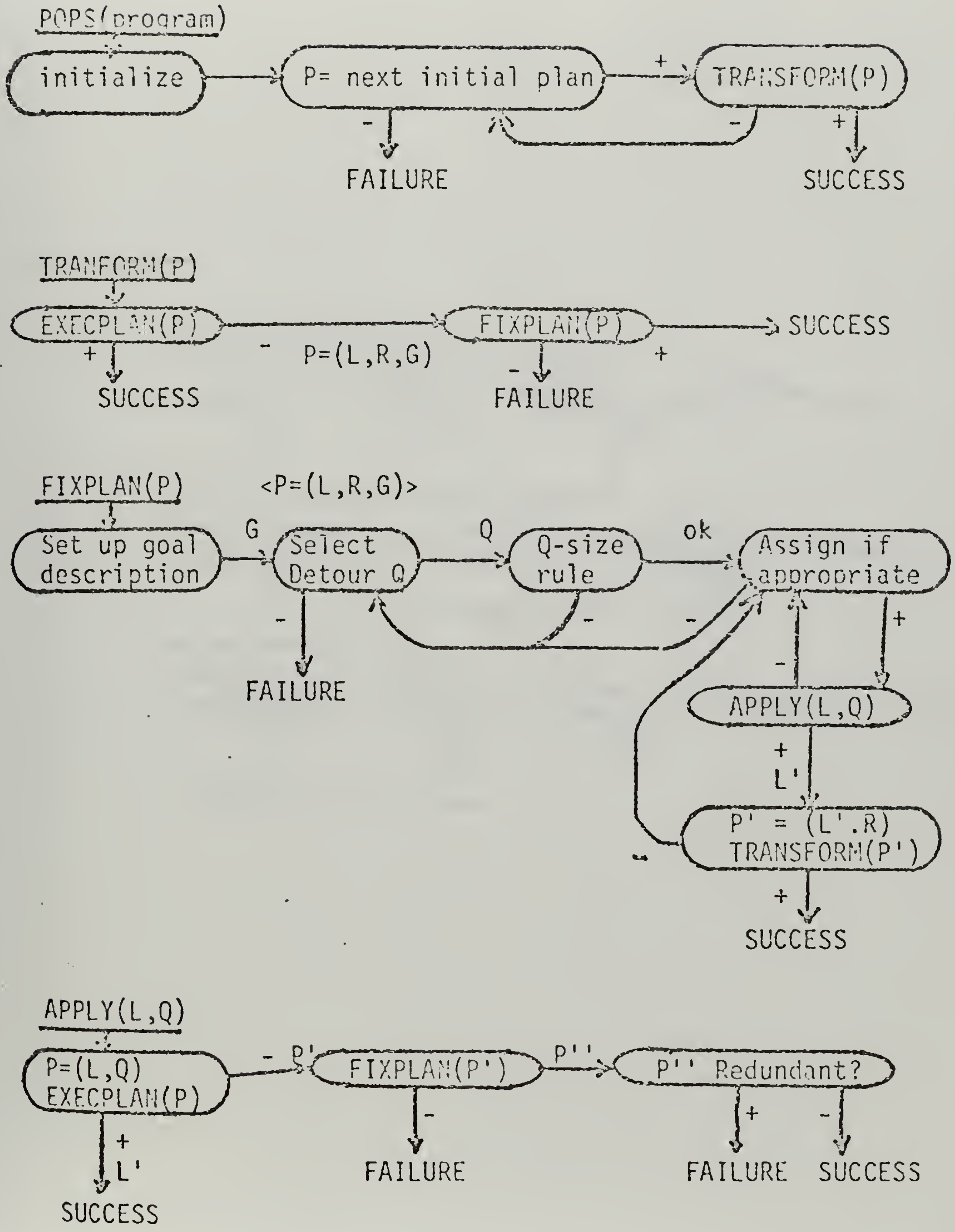
2. POPS Overview

POPS is a program which worked on heuristic search problems using a GPS-like control structure (see Figures 3 and 4) [Newell and Simon 1963]. The object of POPS was to use information found in the problem to control the search. REF-ARF had no directional controls other than the minimum number of possible instructions leading to the END statement. POPS first set up a description of the goal and the operators according to fixed attributes described below. It used these descriptions during the search to choose an operator to add to the current path. Operators were usually one or more PSL statements grouped as a detour, which was a loop in the program, as described below.

3. POPS Initialization

The POPS system started with an initialization section. This section generated a grammar representation of

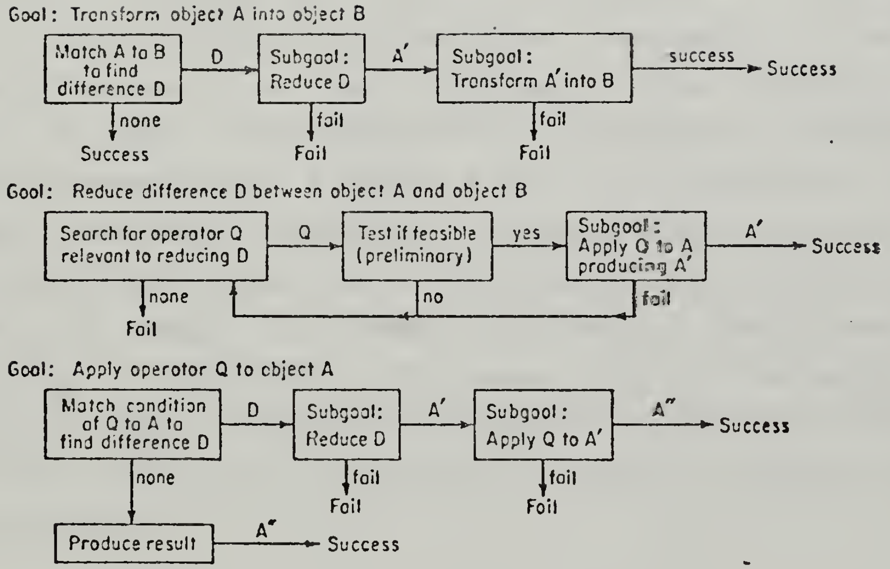
POPS CONTROL STRUCTURE



(from [Gibbons 1972])

Figure 3

GPS CONTROL STRUCTURE



(from [Newell 1963])

Figure 4

the program. This grammar described the possible paths through the program by listing the rules which governed how the labels could be put together as paths. This section also produced a description of the operators, (called detours or moves), their effects, and some initial plans to begin processing.

POPS did an analysis of the structure of the input program and produced a grammar in the following way: first the program was altered to have the label INITIALIZE placed on the first statement and then the program was reduced to using only GOTO, GOTOL, and CONDITION statements to indicate the flow of control in the program. This required the altering of each IF statement to a series of statements. The first statement was a GOTOL with two generated labels. One label was on a CONDITION statement which contained the boolean expression from the IF statement and followed by a GOTO to the label in the IF statement. The other label started with a CONDITION statement which contained the negation of the original expression, followed by the next instructions after the IF statement. This is made clearer with an example:

```
              .
              .
              .
       (IF (= (B2 3) 1) ACT)
              .
              .
              .
```

would become:

```
              .
              .
              .
       (GOTOL (LABEL1 LABEL2))
LABEL1 (CONDITION (= (B2 3) 1))
       (GOTO ACT)
LABEL2 (CONDITION (*NOT (= (B2 3) 1))
              .
              .
              .
```

Also, if the statement following GOTO or GOTOL statements were not labelled, they were labeled. With these modifications the entire program could be divided into moves as defined below.

A move was considered to be a sequence of instructions beginning either with the first instruction or immediately after a branch point and terminating with either the END statement or another branch instruction. These moves were the input to the routine which produced the grammar.

This grammar involved only labels in the program and only showed possible flows of control. The productions in this grammar were of two types. The first type was those possible paths which began with the first label, INITIALIZE, and went directly to the label preceeding the END statement. These direct paths were used as initial plans for the next step in the solution process. The second type of production was a loop which was possible within the program. These were refered to as detours in that they returned control to the first label so that the effect was a detour in the original plan. An example showing both of these types follows.

If the input program was:

```
(     (RANGE A (PT1 PT2 PT3))
      (*SET (M 1) TO PT1)
      (*SET (M 2) TO FL)
A     (GOTOL (WALK HALT))
WALK  (CONDITION (= (M 2) FL))
      (*SET (M 1) TO (SELECT A))
      (GOTO A)
HALT  (CONDITION (= (M 1) PT3))
      (END))
```

the grammar would be:

```
SENTENCE <-- INITIALIZE A HALT     (type 1)
       A <-- A WALK A              (type 2)
```

The object of this analysis was to identify the detours which would be later inserted into an initial plan as specified by the grammar. This process is explained below.

The description of the detours included several attributes considered necessary to decide which detour to apply in searching for a solution. This description was an attempt to describe the effects of the detour. One part of this decription was a list of variables, DEPON, whose value was used before being assigned in the detour. A variable could be included because it was used in a SET statement as part of the value assigned to another variable. Another attribute, CHANGES, was a list of variables whose values were changed by the execution of this detour. There was also PRECOND, a list of conditions which had to be satisfied before this detour could be applied. A logical attribute, ASSIGN, indicated whether this detour required assignments to variables before execution. This was designed to prevent trouble in the evaluation of nested arithmetic expressions by making assignments necessary to avoid the nesting. For example, if the move contained an assignment such as X(1)=X(1)+SELECT(A), and X(1) currently had the value SELECT(A), the number of possible new values appears to be N^2, where N is the size of the range A. But, in fact there are only 2N distinct values. By forcing the assignment of the old SELECT before applying the current move, POPS avoids this redundancy in the processing of constraints.

Finally, there was a numeric value, QSIZE, which was 1-1/(1+N), where N was the number of symbols in the context

resulting from the execution of this detour on a null context. This value was normalized on the interval (0,1) and was used later as a measure of "difficulty" of the detour. These descriptions were grouped as attribute-value pairs in a list. To give an example, the following is the description of the detour A WALK A from the last example problem:

```
(A WALK A) <-- ((ASSIGN)  (QSIZE  .  9.687499E-01)
           (PRECOND  *AND  (*AND  (=  (M  2)  FL)))
           (DEPON)  (CHANGES  (M 1)))
```

The detour is (A WALK A).

ASSIGN is null, thus there were no assignments to any variables necessary.

QSIZE is the measurement described above.

DEPON is null, thus there were no variables used without values by this detour.

CHANGES is the variable changed, (M 1).

PRECOND is the condition which must be met to execute this move, i.e., (M 2) = FL.

4. Execution of a Plan

The grammar represented the way initial plans and future plans could be modified. The detours were loops since they returned to their first label. If a detour looped on any label in a plan, that label could be replaced by the detour according to the grammar. Again using the example problem from above, the initial plan is:

```
(INITIALIZE A HALT)
```

and the detour:

```
A ---> A WALK A
```

they could be used to form a new plan:

```
(INITIALIZE A WALK A HALT)
```

The attempted execution of other than a successful plan resulted in left and right parts of the plan and a constraint. The left part contained the labels which had been executed successfully up to this point. The right part included those labels which had not been executed due to one of three reasons: a false precondition for one of the moves, something evaluated to false within the move, or at the END statement, not finding the solution. The false constraint was the precondition of the first move in the right part and was where execution halted. This expression was used in making the choice of the next detour to attempt to satisfy these conditions.

Using the example problem, the execution of the initial plan, INITIALIZE A HALT, would produce:

```
Left part:    INITIALIZE A
Right part:   HALT
Precondition: (= (M 1) PT3)
```

This constraint was then processed into a description which included two attributes, CHANGES and KEEPS. KEEPS had as its value those slots whose present values were satisfactory for the constraint. CHANGES included those slots whose values needed to be changed to make the constraint true. These two lists were used in the selection cf which detour to apply. The heuristic used to

make this selection was to choose that detour which changed the most variables which needed changing, and also the least of those which did not need changing. The choice was arbitrary for two detours with the same evaluations.

5. Prevention of Redundancy

The redundancy problem [Gibbons 1972,1973] is the waste of generating identical paths in several ways. This greatly expands the effort required to search the search space. The redundancy in the paths attempted was controlled in POPS by two methods. The first method limited the generation of paths so that each possible path was attempted only once in the search. The second method checked for paths which were equivalent in terms of their results. The first method was accomplished through the application of Gibbons' Q-size rule.

The Q-size rule prevented the generation of identical paths by controlling the detours inserted into a plan. Each detour was assigned a value which was part of its description, QSIZE, during the initialization step. These values were used to restrict their detour's introduction into a plan. Formally stated, a detour would be considered only if the following held:

```
if the path was: P=(L,R)
     and D is the detour to be inserted,
         |D| is the Q-size of D,
         |R| is the Q-size of the first label in R,
         |L| is the Q-size of the last label in L,
         and
              |L|≥|D|<|R|.
```

The QSIZE, calculated from the test of the detour as described, was considered to be a measure of the "difficulty" of the detour. The higher the "difficulty" of a detour, the closer the QSIZE for that detour was to one. Thus POPS was designed to use "easier" detours to reach a goal, and detours on the same level must be of equal or decreasing "difficulty". As an example, if the list of numbers, (4 4 2 3 1), represent the QSIZEs, multiplied by ten, for a series of detours in a given path, then they must have been generated as follows:
(the period separates the left and right parts)

```
        4.
      4 4.
      4 4. 3
    4 4 2. 3
  4 4 2 3.
4 4 2 3 1.
```

Using this rule, POPS could be sure of trying each possible plan at most once, without the expensive checking of each path generated with all previous paths. A proof of this is given in [Gibbons 1972].

The second method POPS used to avoid redundancy was the test of path equivalence. Equivalent paths were two nonidentical paths with the same effects. For this test contexts resulting from the execution of left parts of plans were paired with right parts and saved for comparison purposes. For a path to be considered redundant, the right sides had to be the same and the context equivalent to the saved context. The equivalence of contexts was tested by attempting to find a mapping from one context to the other which shows that the values of the slots were identical. Since this test did not consider the constraints in a context, if there was a mapping and the constraints were

different, the node was found redundant when it was not. This caused the missing of a solution in one problem described in [Gibbons 1972].

6. Comments

The POPS system applied heuristic search methods, mainly GPS's, to the design of an interpreter for a nondeterministic programming language. The major innovations in POPS were the descriptions of moves and goals and the application of the Q-size rule. These descriptions allowed much better heuristic selection methods than REF-ARF's count of the minimum possible instructions to the END statement. The Q-size rule eliminated the cumbersome problem of avoiding duplicate paths. With these improvements, POPS was able to solve several problems.

The heuristic which POPS used did not check to see if the detour it selected was going to actually be beneficial, but rather selected from those detours which would make some changes to the variables a detour which did not satisfy the goal with their current values.

To proceed more efficiently toward the solution, the detour selected at the beginning of the branch had to be the one required by the Q-size rule in order to even allow the generation of the path which was the solution. This detour was not necessarily one of the first detours a heuristic might choose. The correct detour selection is discussed in more detail below. This requirement was discovered when trying to improve the detour selection. As an example, in one problem described later the necessary first move, according to the Q-size rule, contradicted the part of the goal which was already true and was hardly the obvious first choice to human problem solvers. The Q-size rule can also block the discovery of a solution at the first opportunity.

As an example, if, in searching for a solution, POPS got within one detour of the solution and was about to choose the correct detour, but the Q-size rule prohibited it, the system never knew it was close. In this case nothing was gained from the effort applied other than the fact that that the solution was not on that branch of the search space. These areas of difficulty are the basis of this study and POPS II.

III. POPS II

A. INTRODUCTION

POPS II is a modified version of Gibbons' POPS. It runs on the NPS LISP [Kennedy 1973] in use at the Naval Postgraduate School, Monterey, Ca. Changes to Gibbons' work include a new method of detour selection, a replacement for the Q-size rule and the ability to re-attack a problem.

B. SELECTION OF MOVES BY REASON

Problem solvers have been choosing moves to apply in a given situation for several reasons. One type of method, such as depth or breadth first algorithms, is fixed in advance and is independent of the problem. Floyd's suggestion is of this type. A second type are those methods which select the move by using a preconceived evaluation of the operators which is dependent on the problems it is specifically designed to solve. An example of this is GPS's table of connections [Newell and Simon 1963]. This table diagrams in advance under what circumstances a specific operator should be applied. When these conditions and operators are studied in depth, the results of the algorithm are very impressive. These reasons are all basically programmed into the problem solver and are unchanged by its operation.

Another alternative is to put off this analysis until the execution of the program. POPS does this by generating a description of its moves at the initialization stage. At the time of this decision of which move to use, it is these

descriptions which are referred to, as opposed to a fixed table similar to GPS's table. This can be carried one step futher by doing most of the evaluation of moves at the actual time of the decision. This could be done by making the static descriptions of the operators at the beginning of the program, and then delaying the evaluation of the operators until the situation in which they will be applied is available. The point is that the value of the move is dependent on the situation in which it could be applied, and that the evaluation should be made without the costs of actually applying the move.

C. LEVEL OF A PROBLEM

The Missionary and Cannibals problem (Figure 5) is difficult to many people because they are using a heuristic such as "increase the number of people on the right side of the river". This leads to problems at one point when the only move possible is apparently "backwards" or against this heuristic. At this point, many people become confused and do not see any way to continue. This move is noted in Figure 6. The "backwardsness" of this move is the basis of the concept of level.

To discuss level there must first be a classification of moves. Let a simple move be a move which consists of the changing of the value of only one variable. Thus general moves are uniquely decomposable into a set of one or more simple moves. The simple moves can be classified into forward, backward and horizontal types at a given node in the search space and for a given goal. The forward classification means that the simple move causes part of the current goal to become true when in the present context it was false. Then a backwards simple move is one which negates a currently true part of the goal. The horizontal

MISSIONARIES AND CANNIBALS PROBLEM

English statement:

There are three missionaries, three cannibals and a boat on the left side of a river. They must cross the river but the boat will only hold two persons. Another problem is that if the cannibals out number the missionaries on either side they will eat them. Thus the problem is for the group to cross the river safely, from the missionaries point of view.

Figure 5

MISSIONARIES AND CANNIBALS SOLUTION

SINGLE MOVES

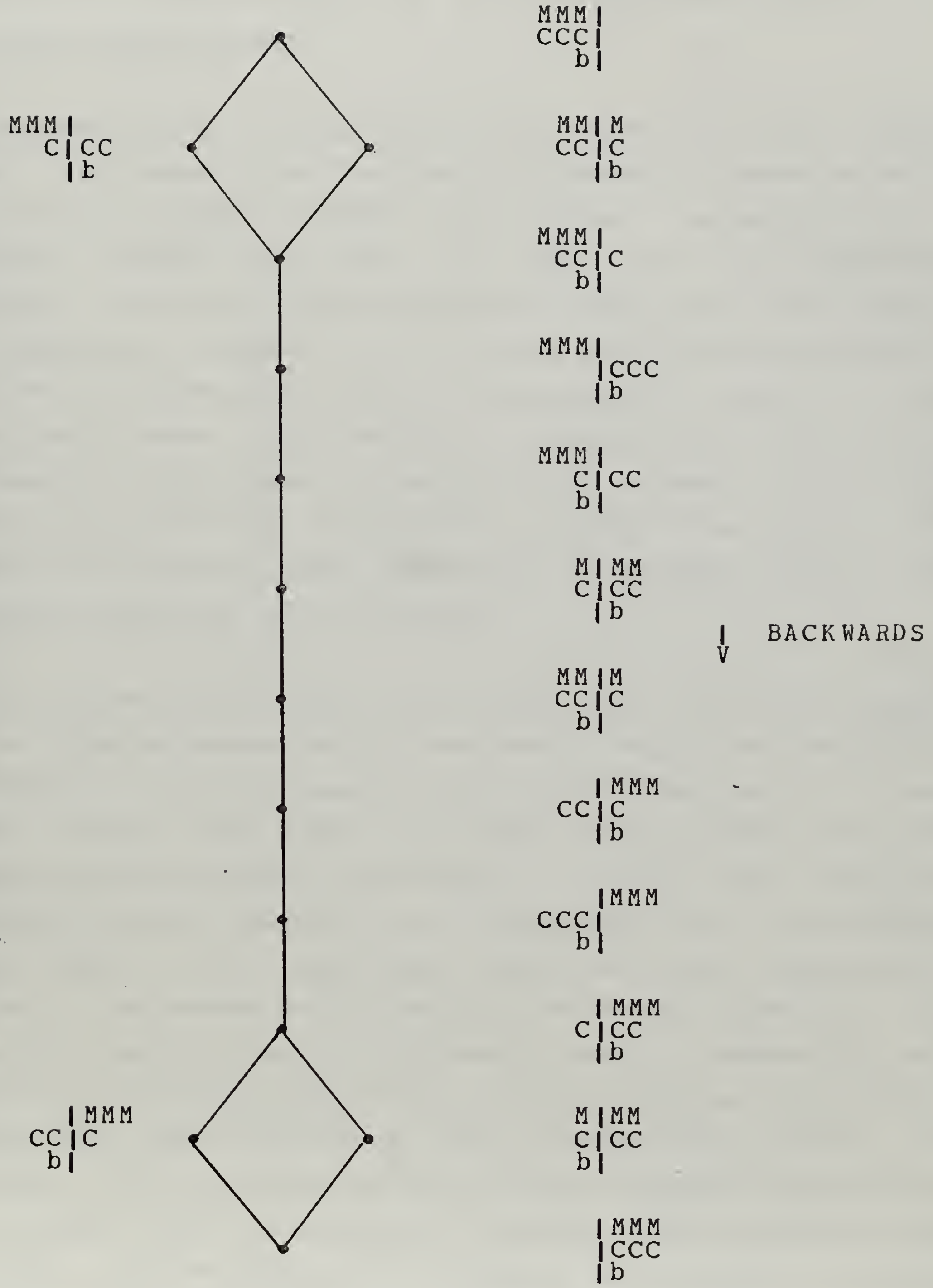


Figure 6

classification is given to a simple move if the variable is not in the goal or if the truth value of no part of the goal is changed by the move. A general move is forward, backwards, or horizontal if the number of its forward simple moves is more than, equal to, or less than the number of its backwards simple moves.

The definition of the level of a problem is the maximum number of consecutive "backwards" (general) moves required to solve the problem. (Note that this classification is different from one which is based on the cumulative backwards "distance" from the goal.) Then the Missionaries and Cannibals problem is a level one problem because it requires one backwards move. A level zero problem is one where each move is forward or at least horizontal. These problems are fairly easy to solve since almost all of the moves are obviously progressing toward the goal. The concept of level has been hidden by two methods which tend to reduce the level of a problem.

The first method which may reduce the level of a problem is to combine moves into a new move. Using the Missionary and Cannibals problem again, one subject considered a move to be across the river and back. The solution for this problem representation is in Figure 7. In this case the backwards move was reduced to a horizontal move, the problem became level zero, and the solution was considerably simpler. The reason this combining of moves worked was that the moves combined were a forward and a backwards move resulting in a horizontal move. If the problem had required a backwards, then horizontal, then forward move series, the combining of two moves would not have reduced the level of the problem. To be sure that a problem would be level zero, it might be necessary to combine many moves, consuming relatively large amounts of resources when compared to the search itself. Unless this combining is directed in some

MISSIONARIES AND CANNIBALS SOLUTION

COMBINED MOVES

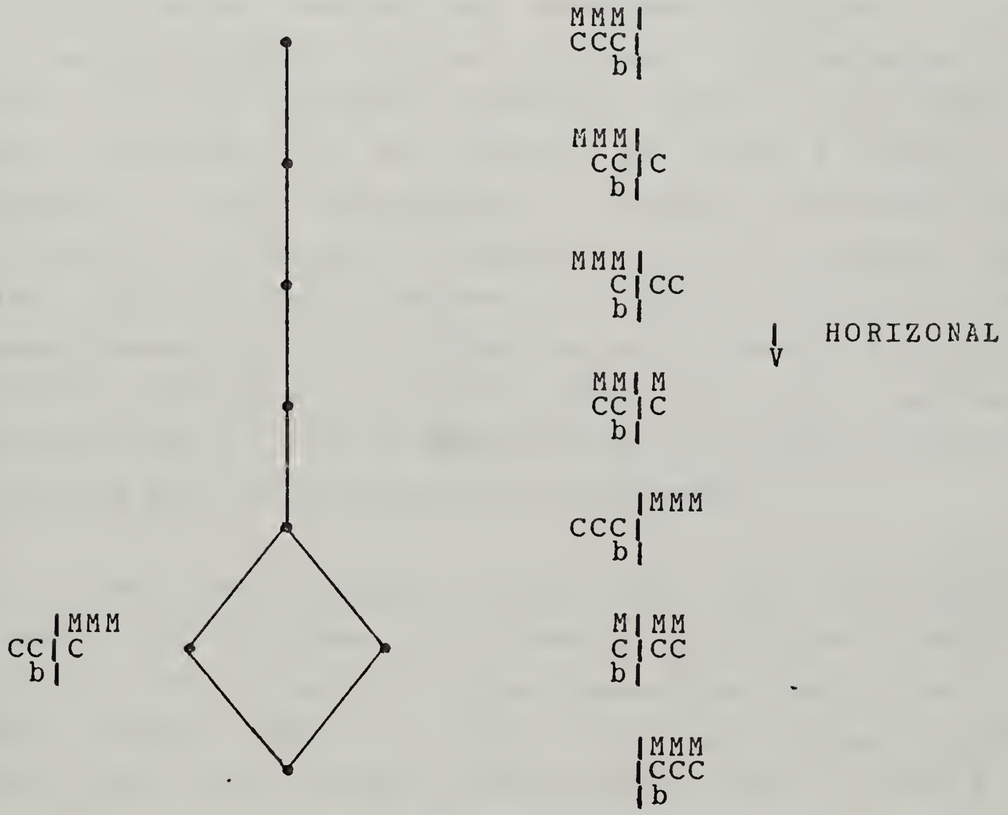


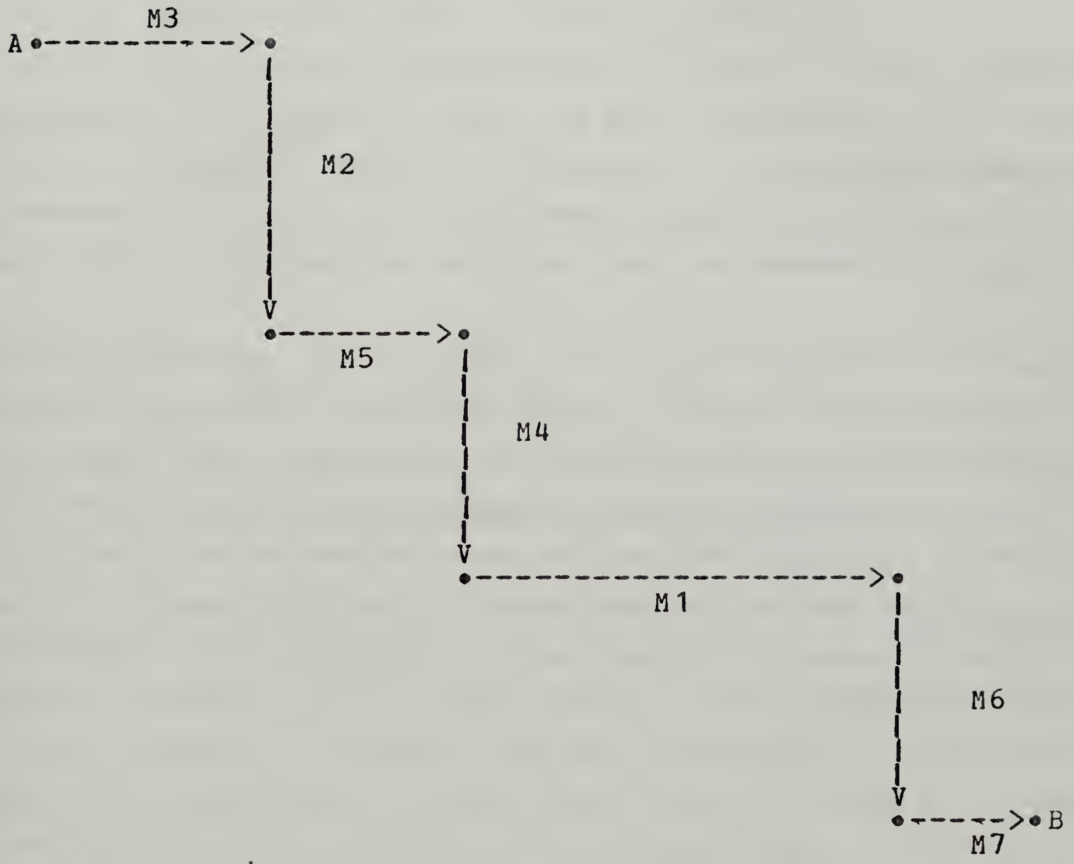
Figure 7

way, it might be necessary to combine more and more moves until the solution is combined into one move and the solution is found in one step. This is exactly what the combining of moves was trying to avoid, a blind search. Thus the combination of moves without direction is no better than a breadth first search.

There is another method of solving problems other than level zero which humans and machines do and/or can use. This is using problem reduction [Nilsson 1971] to attack the problem. If the problem's solution requires more than one move, the problem can be broken up into a series of subproblems. Each subproblem is either involved with solving part of the goal directly or is involved with creating the conditions necessary to apply some other move. The requirements for the application of a move, i.e., the conditions necessary to apply this move can become an intermediate goal. Each subproblem can be reduced until an intermediate goal is attainable in one move.

In a level zero problem the goal and each intermediate goal would be obtained with moves which are forward or horizontal with respect to the preceding move and the following move. Thus if each move is evaluated in the current context and for the current goal, only forward or horizontal moves would be used at each step. Humans use this method when they recognize a key move and then work to apply that move. In this case they are making that move the intermediate goal. This is better explained with a diagram (see Figure 8). The object is to go from point A to point B and the move M1 is recognized as a good move to use. Then the next goal is to get to the point where move M1 can be applied. Then move M2 is applied to get to M1, etc.

PROBLEM REDUCTION PROBLEM SOLVING



Notes:

1. All moves are forward
2. The length of the move is proportional to its value (1 unit = 1 simple forward move)
3. The moves are choosen according to the indicated move number: M1, M2, M3, ...

Figure 8

D. EXPANSION OF THE GOAL

So far the only information received from the problem statement is the information explicitly stated in the problem. This is why there could be simple moves which are considered horizontal when in reality they are not, since the values they change should be in the goal of the problem formulation. It would aid the move selection if it were possible to classify moves as forward or backwards only. The extraction of the implicit data necessary would aid this effort. This information would be used to expand the goal.

The expansion of a goal is the inclusion of more variables with their required values. These values would be those which are mandated by other parts of the original goal. This is distinguished from working backwards from the goal in that the expansion does not derive required previous moves but rather additional requirements for the goal. As an example, in a running of the Monkey and Bananas problem (Figure 9) on POPS, if the goal has as its condition only that the monkey is on the box and the monkey is under the bananas, the solution is found after about 28 seconds. When the goal is expanded to include the requirement that the box is also under the bananas the solution requires only 18 seconds. Finally, if the fact that the box is on the floor is added, the solution required 18 seconds again because the information could not be used since the variable's value could not be changed anyway.

Although this looks like it would be very worth while, the deductive power required appears prohibitive. What is given is a partial description of a node in the search space and what is desired is as complete a description of the node as is possible. This can only be done by analysis of the

MONKEY AND BANANAS PROBLEM

English statement:

In a room are a monkey, a box and some bananas hanging from the ceiling. The monkey wants to eat the bananas, but he cannot reach them unless he is under them and standing on the box.

Moves:

WALK - changes the monkey's position
CARRY - changes the monkey's and box's position
CLIMB - changes the monkey from the floor to the box or vice versa

Conditions for Solution:

1. The monkey is under the bananas and standing on the box.
2. The above and the box is under the bananas.
3. The above and the box is on the floor.

Figure 9

moves in the problem to see how they effect the values of variables. An example of this in the Monkey and Bananas problem is the discovery that for the monkey's position to be changed, it must be on the floor because the only moves which change the monkey's position are the walking and carrying moves.

E. CONTROLLING SEARCH WITH CUTOFF

For the ideal problem solver, the next move is clearly defined to be only one move at each step in the search. This implies that the search space has been narrowed to only one move at each node. This is what has been described as expertise in human problem solvers [Gibbons 1972]. The next thing to do is "obvious" and either all other possibilities are ignored, or the thing to do next is selected from a narrow subset of the moves which are avaiable. A heuristic which could be used would be to choose that move which is the most forward or at least horizontal. This heuristic assumes that the most promising or best move is probably going to be the move which makes the most immediate progress toward the goal, and which therefore must be a forward or horizontal move as described above.

To narrow the search space and limit the wanderings of the problem solver, a limit on the reason for choosing a move seems reasonable. That means the system is prevented from picking just any move, but is required to choose a move from those which make progress toward the goal. This is done by the means of a cutoff. If the cutoff is set to only allow forward moves, then all other moves are barred from selection and so forth. In the limit where the cutoff allows all moves, the problem solver wanders without control. The acceptance of forward and horizontal moves type allows for the inaccurate evaluation of some moves

which were mistakenly classified as horizontal.

The use of the Q-size rule has also been described to be a method of controlling the search. However, there are problems involved in the Q-size rule. If the problem solver gets to a node, and the application of one more move actually obtains the goal, it can be blocked because it violates the rule. Thus the Q-size rule helps avoid redundancy but places restrictions on the possible approaches to the solution.

F. CONTROLLING REDUNDANCY

The Q-size rule's application allows only one method of generating a path to a state. Unfortunately this rule conflicts with heuristic search methods. Any attempt to allow the free use of a heuristic within the Q-size's search space, would break through the rule's redundancy protection.

There appear to be two methods to improve the Q-size rule, but they do not eliminate the problem. The first way would be to get a better measurement of the "difficulty". Difficulty, even an accurate measurement of difficulty, is not directly related to the usefulness or applicability of a move in a particular situation. Thus, a heuristic which is trying to choose moves according to their ability to reach or move toward the goal will be restricted arbitrarily by this Q-size rule. The second method to improve the rule would be to use some other measurement than "difficulty" as the QSIZE of a move. This would not work either because the problem with the Q-size rule is the rule itself. The Q-size rule needs an ordering of the moves at the begining of the problem, or at least a fixed ordering determined shortly after the start of the problem. What is desired is that the rule not interfere with the heuristic search except in the

generation of identical states. This is not possible unless the ordering is dependent on both the heuristic used and the location of the goal in the space. At the begining of a problem, this information is unknown and therefore the Q-size rule cannot have the necessary ordering of the moves, even assuming that there is one. Using the rule, it could be possible to use the path to return to the same point following the Q-size rule.

One further point; the Q-size rule was designed to incorporate the idea that people never attempt subproblems that are more difficult then the original problem. The formalization of this point has been shown to be confining to an heuristic search. Therefore it seems reasonable to conclude that the observation was not accurate enough to assume that more difficult subproblems are never attempted. It is not until a person is working on or has completed a subproblem, that a measurement of its difficulty can be given. Thus, the formalization of this observation into a rule was unnecessary.

Without the Q-size rule the redundancy problem remains. The following is a proposal which is not as elegant or as inexpensive as the Q-size rule, but should allow any heuristic search method. This method is simular to the hashing [Gries 1971] used in organization of symbol tables in compiler design. A hashing function maps the input domain into an index of a linear array. As long as two elements in the input set do not map into the same index, the cost of the searching is only that of the hashing function. In order to use this to prevent redundancy, the only entry in the linear array would be a flag to indicate if an input, representing a state in the space, had already been hashed to this index before. This is under the assumption that there will be at most one state mapping into each element in the array.

The following is a specific proposal which could be used in a system. Since a path defines a state in the search space, paths will be used as the input to the hashing function. For each move applicable in the problem, assign a number starting with zero. The hashing function which would map each possible path to a unique index is:

$$I = M_1 * N^0 + M_2 * N^1 + M_3 * N^2 + \ldots + M_k * N^{k-1}$$

I - the index

N - total number of moves

M_i - move number of the ith move in the path

k - number of moves in the path to be hashed

The array could be in bits, where for a given maximum depth, the number of bits necessary would be:

$$B = N^d - 1$$

B - bits necessary (also total possible paths)

N - number of moves in the problem

d - maximum depth of the search

Since this function is 1:1, one bit is used to represent a path. On small problems this fuuction can be very practical. As an example, if there were 5 moves, and the maximum depth to be searched was 5, then it would require 3,124 bits, or about 100 32-bit words. For larger problems, a hashing function which has a smaller range than the total number of paths possible might be used. To use a smaller range requires measures to deal with collisions and this is usually done with more programming than the hashing function alone. Although this would cost more in time, the storage requirements can be reasonable.

In the proposed hashing function zero would indicate that the path had not been previously checked and one would indicate the path hashed was redundant. The array would be initialized to all zeros. Each hashing would either set the bit indexed to one, or discover that the path had been attempted before. This method of checking for redundancy would cost more than the Q-size rule did, but it would not interfere with the generation of the paths; and therefore, could be used with any heuristic search method.

G. RECOURSES FROM FAILURE

When an heuristic search problem solver fails to find a solution, because the search was too limited, there are methods to re-attack the problem. These recourses depend on the type of search controls used. If the nodes at which the search was halted were saved, then the search can be reinitiated at the promissing nodes. In the case of a processor using cutoff, if the first search was limited to "good reasons" or forward moves only, this can be restarted by lowering the cutoff to allow horizonal moves also.

For any processor, if there is a learning process as the search progresses, the solution might be more accessible with this knowledge on a second try. This learning might include information as to what moves, or combinations of moves, do and do not work to obtain certain goals. Upon the completion of a search of a branch, the processor should learn more than that the branch does not contain the goal. This information could be used by a processor using the Q-size rule, at the top, when the access to a goal is blocked by the rule.

H. THE POPS II PROGRAM

Several of the above suggestions have been implemented in POPS II. POPS II chooses its detours by their classification of forward, horizontal or backwards. It also uses the cutoff mechanism to control the search and has a multipass capability using various specified values for the cutoff.

Each detour is tested to get an overall classification of its type. This is done not by the execution of the detour but by the modification of the detour's description. The description in POPS had contained the variables which were changed by this detour. This was modified to include the values to which those variables were changed. Nondeterministic assignments are indicated with their ranges' name. Each detour is tested in turn and its effects are evaluated as to whether it makes part of the goal true, could make part of the goal true (in the nondeterministic case), has no effect, or negates part of the goal. The test is a search for variables both changed by the detour and in the goal. Then a trial assignment is made and the results are evaluated according to whether the assignment makes a false part of the goal ture, a true part false, or makes no change to that part's value. The sum of these evaluations, with a value for those which are changed but are not in the goal, and a weighting of the QSIZE of the detour, is used as a net value of the detour. The detour with the highest value is then selected for application, provided it is above the cutoff. In the examples discussed, the weighting of assignments which actually and nondeterministicly made part of the goal true was equal and opposite in sign to the weighting given assignments which negated part of the goal. The QSIZE was not included in the evaluation of a detour.

The cutoff mechanism is a test of the net value of the selected detour to see if it is above the threshhold value which was previously specified. If it is, the detour is passed to the main program. If it is not, no detour is passed on thereby indicating the end of the search of this branch. At nodes where the search was cut off, the plan is saved as a future plan to be used in the next pass.

The multipass capability is implemented by saving the future plans and necessary data and by starting the process over upon failure of the previous pass. To start the next pass, the cutoff is changed to its next value and the saved future plans become the new initial plans. This was found to require much more time than a single pass with a lower cutoff.

The output of POPS II includes information on the initialization and search carried out. The source listing, moves as the program sees them, the grammar, and the detours are printed out before any searching is done. The search starts with a list of plans which it will try and then the execution and modifications to those plans. The system also traces itself through the flowchart (Figure 10) during the search. At the end of one pass, the paths considered are listed in reverse order and the future plans are also listed.

I. POPS II PERFORMANCE

The performance of POPS II is described by examining its execution of several sample problems. The search done by POPS and POPS II are compared in terms of nodes searched and time spent on the problem. A comaprison of POPS and POPS II is in Table I. No multipass examples are included due to the fact that a single pass with a wider search space took

POPS II CONTROL STRUCTURE

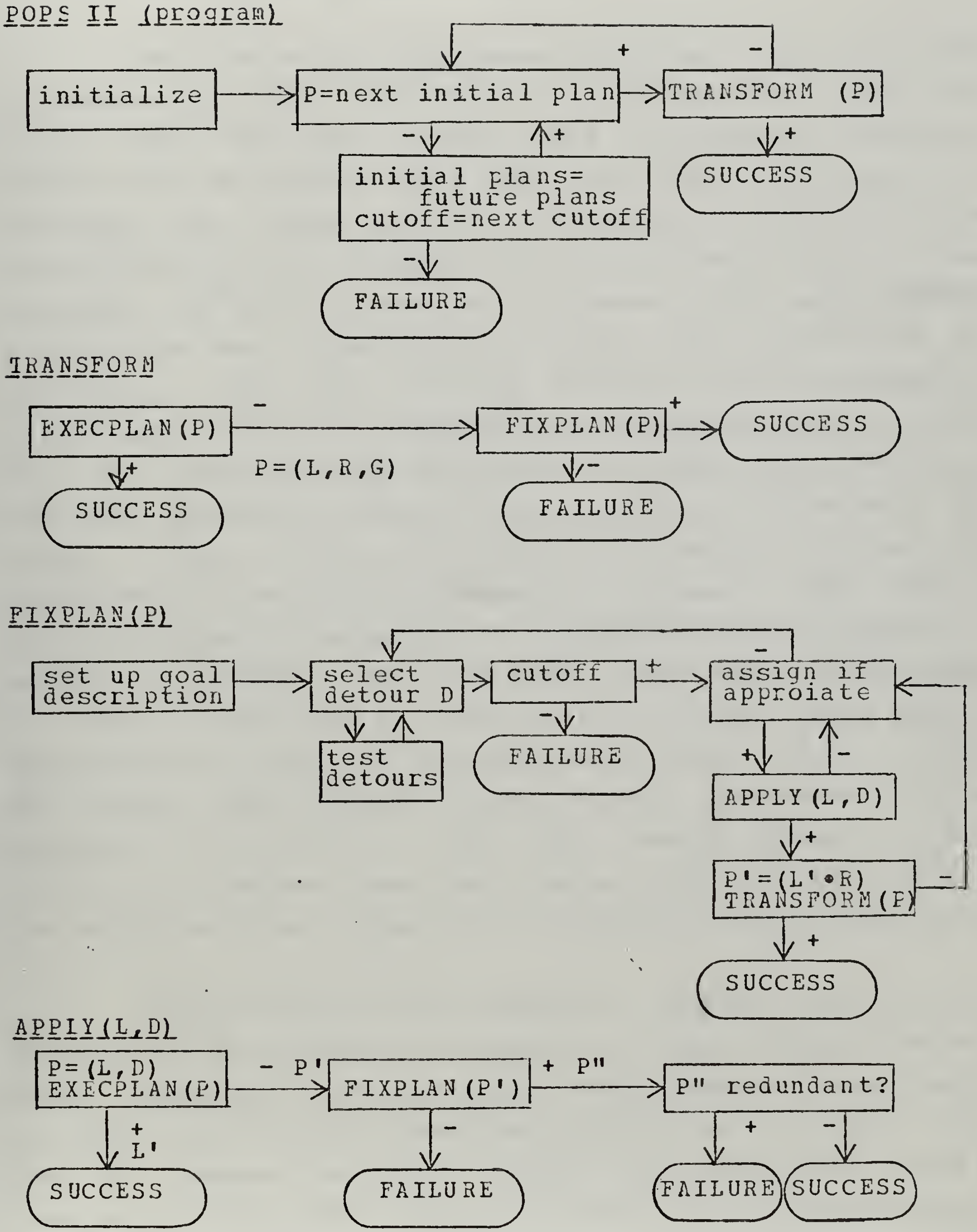


Figure 10

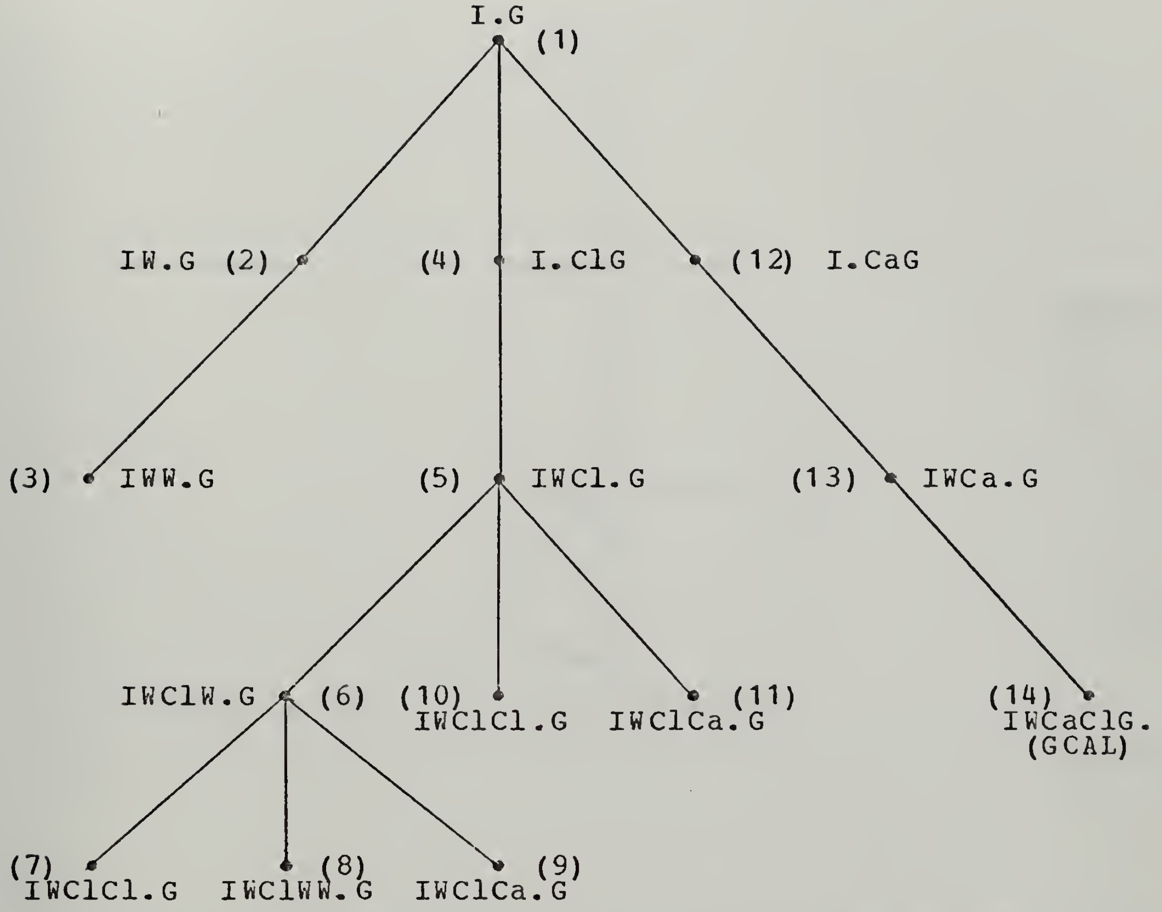
much less time than the multipass runs.

1. Monkey and Bananas Problems

On the Monkey and Bananas problem, (Figure 8), POPS II behaves very much like POPS. On this problem, with the first goal statement, namely that the monkey is under the bananas and on the box, both POPS and POPS II carry out basically the same search. A diagram of the search tree carried out by both problem solvers is Figure 11. In the analysis of moves which POPS II makes, WALK and CLIMB had exactly the same value so their order of application was arbitrary. For both problem solvers, the move CARRY was tried last because it apparently had the same effect on the goal as the move WALK, and since it changed more variables than was apparently needed it was considered after WALK and CLIMB. POPS II went farther on the branch starting with CLIMB because, in the evaluation of progress, made when a nondeterministic assignment is the basis of the progress, no advance is noticed. Thus the WALK, then CLIMB, moves (node 5) could be followed by another WALK and CLIMB move pair in attempting to reach the original goal (nodes 6,7,8, and 10). The search was blocked from further expansion by the discovery of redundancy. What this search shows is that without a complete goal the problem solvers will only find the solution after trying all the obvious moves first.

When the goal was changed to include the fact that the box had to be under the bananas, both problem solvers went directly to the solution. The diagram of this search is Figure 12. POPS II deviated slightly to try CARRY again for the reason explained above. This diversion would be removed if the actual reaching of the goal was valued higher than two nondeterministic assignments. The addition of the last item of information, that the box had to be on the floor, did not effect either problem solver. This

MONKEY AND BANANAS (1) SOLUTION TREE



PATHS:
I = INITIALIZE
W = WALK
Ca = CARRY
Cl = CLIMB
G = GOAL

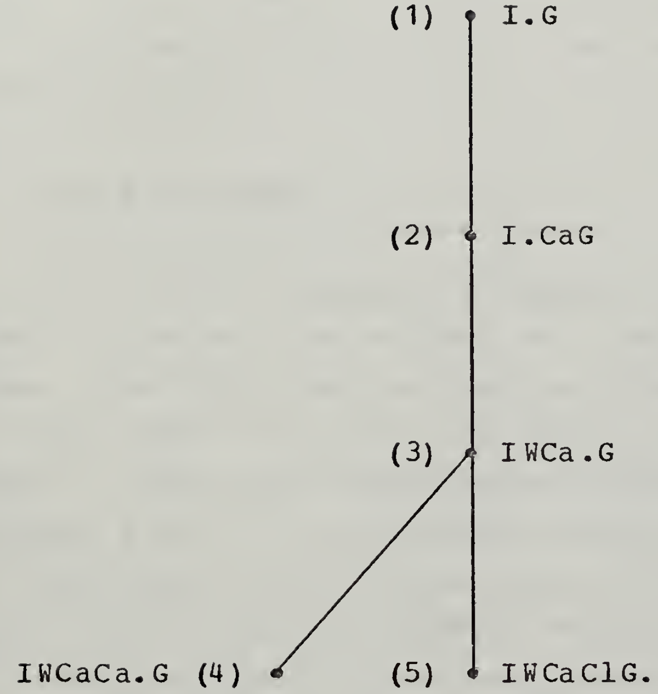
GOAL: The monkey under the bananas and on the box.

SEARCH: POPS: 1,2,3,4,5,6,10,12,13,14

POPS II: 1,4,5,10,6,7,8,9,11,2,12,13,14

Figure 11

MONKEY AND BANANAS (2,3) SOLUTION TREE



GOAL: (1) The monkey is under the bananas and on the box and the box is under the bananas.

(2) The above and the box is on the floor.

SEARCH: POPS: 1,2,3,5

POPS II: 1,2,3,4,5

Figure 12

information was actually useless since none of the moves changed the height variable of the box.

These problems serve to show the effect on both POPS and POPS II of the information contained in the goal. If there is not enough information, the search will not be directed very well.

2. Robot Problems

The Robot problems [Gibbons 1972], Figure 13, are designed to show the effect of increasingly difficult problems. The first task is null in order to get the set up time. With tasks 2 to 5 the problems become increasingly difficult. POPS and POPS II went directly to the solution on problems 2 and 3. On problems 4 and 5, POPS inserted an extra step which had no effect when it was finally assigned a value to its nondeterministic variable. On problem 4 after first deciding to push both boxes, it decided to push one box first. POPS later decided that this box would be pushed to where it was originally, i.e., not pushed at all. The effect of this move is the same as the move WALK. This same diversion was taken by POPS again on the fifth problem. POPS II solved both of these problems without the insertion of null moves. The results of POPS and POPS II on these problems are in Table I.

3. Waterjug Problem

The next problem is the Waterjug problem, described in Figure 14. Although POPS II was able to solve this problem where POPS could not, it cannot be credited to POPS II. The major difficulty is that there are practically no goals to choose detours for in this problem. This means that almost all detours evaluate to horizontal moves for almost all goals. Second, the detours which assign

THE ROBOT PROBLEMS 1-5

English statement:

> In a room are a robot and two boxes, B1 and B2. The robot is at position A and there are four positions in the room, A,B,C, and D. Both boxes are on the floor at position B. The robot can walk from any position to any position. It can also push b1 to any position. It can stack B2 on B1 and then push both boxes to any position. Finally it can remove B2 from on top of B1.

TASKS:

(RB1) Do nothing.
(RB2) Move itself to location C.
(RB3) Move B1 to location C.
(RB4) Move B2 to location C.
(RB5) Move B2 to location C, B1 to location E.

Figure 13

WATERJUG PROBLEM

English statement:

There are an eight and a five gallon container, a water source and a sink. The object is to get two gallons of water in the five gallon container.

Moves:

FILLA - fill container A with 8 gals.
FILLB - fill container B with 5 gals.
MTA - empty container A
MTB - empty container B
PAB - pour from container A to container B until A is empty or B is full
PBA - pour from container B to container A (like PAB)

Figure 14

arithmetic functions to the variables, PAB and PBA, are not within the evaluation capabilities of the processor. Thus the value assigned to the variables is unknown and any evaluation results in a horizonal rating. The additicn of the fact that, at the end, (A 1) = 8, confuses the search because then the FILLA move is forward at the begining and points POPS II off in the wrong direction at the start. With this formulation of the goal, POPS II was not able to solve the problem in over 90 seconds. This problem shows the effect of a problem where there is not a meaningful measurement of move type, and also shows the lack of formula manipulation used in getting the description of the moves.

4. Expanded Monkey and Bananas Problem

The last problem to be considered is an expanded Monkey and Bananas problem, Figure 15. In this problem the move CARRY of the original problem is broken up into three moves, PICKUP, CARRY, and PLACE. The move CLIMB is also divided into CLUP (climb up) and CLDWN (climb down). This problem was designed to make the solution a little more detailed.

The performance of POPS and POPS II varied greatly on this problem. The POPS system searched over 70 nodes in approximately 200 seconds and did not find the sclution. POPS II solved the problem requiring 6 nodes, after generating a total of 9 nodes, with a total time of 56 seconds. Although both processors start with the same move, CARRY, POPS would have to go through 43 nodes before it would start with the move PICKUP, as required by the Q-size rule. The search done by POPS II deviated twice from the direct path to the solution because it was trying to obtain part of its intermediate goal which did not evaluate to true due to the undecided values of some nondeterministic assignments as explained before. This problem shows

EXPANDED MONKEY AND BANANAS PROBLEM

English statement:

In a room is a monkey, a box and some bananas hanging from the ceiling. The monkey wants to eat the bananas, but he cannot reach them unless he is under them, on the box, and his hands are empty. The box must also be on the floor and under the bananas.

Moves:

WALK - Changes the monkey's position

CARRY - Changes the monkey's and the box's positions, if the monkey is holding the box.

PICKUP - Changes the box's location to up and fills the monkey's hands

PLACE - Changes the box's lacation to the floor and emptys the monkey's hands.

CLUP - The monkey's location becomes on the box.

CLDWN - The monkey's location becomes the floor

Figure 15

TABLE of COMPARISONS

PROBLEM	NODES in SOLUTION	POPS		POPS II	
		nodes	time	nodes	time
MB(1)	3	13	28.8	11	38.1
MB(2)	3	3	18.1	4	23.3
MB(3)	3	3	18.1	4	23.3
MB(E)	6	73	>200	9	56.9
WJ	4	43	85.7	19	47.0
RB1	0	0	9.9	0	9.6
RB2	1	1	12.4	1	11.2
RB3	2	2	16.8	2	17.4
RB4	3	4	27.7	3	21.6
RB5	5	7	45.0	6	37.6

Table I

dramatically the conflict between choosing the move which can be shown to be forward and the limitations inflicted by the Q-size rule.

H. FUTURE WORK

There is still very much work to be done in problem solving and problem solving with nondeterministic programming languages. In general problem solving, more work is needed in two areas. First, work is needed in the area of obtaining more information from a problem statement. Second, work is needed on the study of problems in general. The discussion of level is along this line. If methods to reduce the level of a problem can be described in detail, then human and machine problem solvers would profit.

In the area of nondeterministic programming languages, POPS and POPS II are designed to solve problems written as basically heuristic search problems. Gibbons [Gibbons 1972] gave an algorithm for converting general problem statements into heuristic search problem statements. Neither POPS or POPS II has this generality. More work also needs to be done on formula manipulation techniques as in [Fikes 1968]. With these techniques, a more accurate evaluation of move types could be made which would probably aid in heuristic search methods. Finally, research is needed in the protection against redundancy. It is expected that a redundancy check less expensive in time and space can be found to replace the hashing function described herein.

IV. CONCLUSIONS

The conclusions of this study deal with general problem solving and the use of nondeterministic programming languages in problem solving. These conclusions are based on observations and work done and are explained in the preceeding chapters.

Progress has been made in the study of problems and their solutions. First, the requirements for the solution of a problem are often incomplete and can be expanded to allow an easier solution process. Next, in a given situation and for a given goal, moves in the problem can be classified into forward, horizontal and backwards types based on their mixture of simple moves. Finally, problems themselves can be classified according to their level as previously defined.

Choosing moves according to their type, and the application of the Q-size rule, present a conflict in POPS. POPS II uses this method of choosing moves for application, however, it does not use the Q-size rule and is unprotected from the redundancy problem. The use of hashing to prevent redundancy could easily be incorporated into POPS II, solving this problem.

Nondeterministic programming has many advantages for the study of problem solving. First is that it allows a fairly simple method of presenting a wide range of problems to the computer. Second, it allows decisions as to the values of variables and selection of moves to be put off until they are necessary, thus avoiding as much enumeration as possible. For these reasons it is expected that

nondeterministic programming will eventually be used to reach the goals of problem solving.

BIBLIOGRAPHY

Bolce, J. F., "LISP/360" University of Waterloo, Waterloo, Ontario, Canada, 1967.

Feigenbaum, E. and J. Feldman (eds.), *Computers and Thought*, McGraw Hill, 1963.

Fikes, R. E., *A Heuristic Program for Solving Problems Stated as Nondeterministic Procedures*, Doctoral Thesis, Carnegie-Mellon University, 1968.

-----, "REF-ARF: A System for Solving Problems Stated as Procedures", J. Art. Intel., 1(1), 1970.

Floyd, R., "Nondeterministic Algorithms", J. ACM 14(4), 1967.

Gibbons, G. D., *Beyond REF-ARF: Toward an Intelligent Processor for a Nondeterministic Programming Language*, Doctoral Thesis, Carnegie-Mellon University, 1972.

-----, "POPS: An Application of Heuristic Search Methods to the Processing of a Nondeterministic Programming Language", draft submitted to the Third International Joint Conference on Artificial Intelligence, Stanford University, 1973.

Greis, D., *Compiler Construction for Digital Computers*, John Wiley & Sons, Inc., 1971.

Hopcroft, J. E. and J. O. Ullman, *Formal Languages and their Relation to Automata*, Addison-Wesley, 1969.

Kennedy, W. G., "NPS LISP", paper and program contributed to the W. C. Church Computer Center, Naval Postgraduate School, April, 1973.

Knuth, D. E., *The Art of Computer Programming*, vol. 1, Addison-Wesley, 1968.

Newell, A. and Simon, H. A., "GPS, A Program that Simulates Human Thought", in Feigenbaum and Feldman, 1963.

Nilsson, N. J., *Problem-Solving Methods in Artificial Intelligence*, McGraw-Hill, 1971.

Prichard, M. A., "LISP/PAREN", contributed program, Penn. State University, 1969.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexanderia, Virginia 22312	2
2.	Library, Code 0212 Naval Postgraduate School Monterey, California 93940	2
3.	Asst Professor G. D. Gibbons, Code 53Gp Computer Science Group Naval Postgraduate School Monterey, California 93940	1
4.	ENS William George Kennedy, USN RR2. Box 114 Crystal River, Florida 32629	1
5.	Chairman, Computer Science Group, Code 72 Naval Postgraduate School Monterey, California 93940	1
6.	Asst Professor V. M. Powers, Code 52Pw Computer Science Group Naval Postgraduate School Monterey, California 93940	1

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

ORIGINATING ACTIVITY *(Corporate author)*

Naval Postgraduate School
Monterey, California 93940

2a. REPORT SECURITY CLASSIFICATION

Unclassified

2b. GROUP

REPORT TITLE

Problem Solving and Nondeterministic Programming Systems

DESCRIPTIVE NOTES *(Type of report and, inclusive dates)*

Master's Thesis; June, 1973

AUTHOR(S) *(First name, middle initial, last name)*

William George Kennedy

REPORT DATE

June, 1973

7a. TOTAL NO. OF PAGES

71

7b. NO. OF REFS

14

. CONTRACT OR GRANT NO.

9a. ORIGINATOR'S REPORT NUMBER(S)

. PROJECT NO.

.

9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

.

. DISTRIBUTION STATEMENT

Approved for public release; distribution unlimited.

. SUPPLEMENTARY NOTES

12. SPONSORING MILITARY ACTIVITY

Naval Postgraduate School
Monterey, California 93940

. ABSTRACT

This paper suggests and discusses an implementation of several ideas in the area of problem solving and nondeterministic programming systems. After discussing the history of work in this field, definitions of forward, horizontal, and backwards simple moves are given. With these definitions, the level of a problem is defined to be the maximum number of consecutive backwards moves required to solve the problem. Level zero problems can be solved using forward and horizontal moves only. It is then shown how two methods, the combination of moves and problem reduction, sometimes reduce the level of a problem. The use of Gibbons' Q-size rule to prevent redundancy is shown to sometimes conflict with heuristic search methods. The use of a hashing function to detect redundancy is discussed. The choosing of moves according to their evaluation of move types was used in a program, POPS II, based on Gibbons' POPS, which was shown to be more efficient than previous nondeterministic problem solvers for several problems.

KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
problem solving, nondeterministic programming systems, nondeterministic finite automata, Q-size rule, REF-ARF, POPS, level of a problem						

19 JAN 76 23499

Thesis
K3835 Kennedy 144242
c.1
Problem solving and nondeterministic programming systems.

19 JAN 76 23499

Thesis
K3835 Kennedy 144242
c.1
Problem solving and nondeterministic programming systems.

thesK3835
Problem solving and nondeterministic pro
3 2768 002 12110 5
DUDLEY KNOX LIBRARY